IMPLEMENTACJA LISTY

Poniższy dokument pokazuje przykład implementacji PROSTEJ listy, która została omówiona w materiałach dydaktycznych do przedmiotu.

Strony zostały podzielone na dwie kolumny. W lewej znajdziecie Państwo kod programu, nastomiast w prawej komentarze do programu – które opisują implementację.

Podobną listę należy zaimplementować aby rozwiązać „ZADANIE NR 3". Proszę jednak zwrócić uwagę, że poniższa implementacja nie jest idealna! ;)

- Wykorzystuje zmienną globalną i nie pozwala na utworzenie więcej niż jednej listy. Należałoby tak zmienić implementację aby funkcja **initList** zwracała wskaźnik do nowo utworzonej listy. Np.:

```
int main(void)
{
   List *pPietroPierwsze = initList();
   List *pPietroDrugie = initList();

   //operacje na listach

   deinitList(pPietroPierwsze);
   deinitList(pPietroDrugie);
}
```

- Możliwy jest wyciek pamięci
Należy zmienić implementację listy tak aby wywołanie funkcji deinit, gdy lista zawiera jakieś elementy, nie spowodował wycieku pamięci.

- Alokacja elementu pRoot nie jest konieczna.
Przedstawiona implementacja zakłada że **pRoot** jest zawsze utworzony (Alokujemy pamięć dla tego elementu w initList). Możemy jednak skorzystać ze wskaźnika, który jest równy NULL gdy lista jest pusta. A pierwszy element tworzyć dopiero przy pierwszym wywołaniu funkcji **addCar**.

(...)

Proszę skorzystać z tego dokumentu jako z podpowiedzi, a nie gotowego rozwiązania. Zachęcam również do „zabawy" z programowaniem oraz implementację większej ilości funkcji : np. wyszukaj element, zamień dane w elemencie listy już istniejącym; lub implementację funkcji do „obsługi parkingu" np. oblicz czas postoju pojazdu, oblicz kwotę do zapłaty, wyświetl czas wjazdu, wyjazdu w formie przyjaznej dla człowieka itp. itd.
Można również zaimplementować listę dwukierunkową jako rozwiązanie Zadania nr 3.

Czy jest to lista FIFO, LIFO? W internecje znajdziecie Państwo informacje o różnych typach list. Jest to jednak temat dla innego przedmiotu który wykracza poza zakres podstaw programowania.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>

typedef struct List {
   char *pNumRej;
   time_t entryTime;
   struct List *pNext;
} List;

List *pRoot = NULL;
```

COMMENTS:

Deklarujemy listę. Wykorzystujemy słowo kluczowe **typedef**. Zgodnie z jego definicją (patrz materiały dydaktyczne) nadajemy nową nazwę zdefiniowanej tutaj strukturze List {(...)}.

Tworzymy zmienną globalną pRoot. Jest to wskaźnik na naszą listę. Dzięki tej zmiennej wszystki funkcje mogą korzystać z tej listy. Pod tym wskaźnikiem zapiszemy pierwszy element naszej listy. Kolejne elementy będą znajdowały się w „łańcuchu".

```
void initList(void)
{
   pRoot = malloc(sizeof(*pRoot));
   pRoot->pNumRej = malloc(strlen("Root") + 1u);
   strcpy(pRoot->pNumRej, "Root");
   time(&(pRoot->entryTime));
   pRoot->pNext=NULL;
}
```

Tworzymy funkcję inicjalizującą naszą listę. Użytkownik naszej listy (MY, bądź inny programista korzystający z naszej listy) musi wywołać tą funkcję aby „zainicjalizować" listę.
W tej funkcji zaalokujemy pamięć dla pierwszego elementu naszej listy „root". Proszę zwrócić uwagę na:
- strlen (string.h) zwraca długość ciągu znaków bez znaka końca tekstu (NULL na końcu!). Dlatego dodajemu tutaj +1u.

-strcpy (string.h) kopiuje tablice znaków. W tym przykładzie skopiujemy 'Root" pod adres wskazany przez wskaźnik pRoot-> pNumRej.

-time (time.h) – kopiuje aktualny czas systemowy pod adres wkazany przez wskaźnik pRoot->entryTime.
Kopiowany czas jest w formacie „timestamp" czyli liczby sekund które upłynęły od 01/01/1970 roku.

-Kolejny element listy nie istnieje, dlatego wpisujemy pod wskaźnik na niego wskazujący NULL.

```
void deinitList(void)
{
   free(pRoot->pNumRej);
   free(pRoot);
}
```

Tworzymy funkcję która „deinicjalizuje" naszą listę. W tym wypadku programista korzystający z naszej listy musi wywołać tą listę gdy już jej nie będzie więcej używał, albo na koniec programu. W funkcji tej zwalniamy zaalokowaną wcześniej w funkcji init pamięć.
Co się stanie gdy wywołamy funkcję **deinitList** kiedy lista będzie zawierała jakieś elementy?

```
void addCar(char *pNumRej)
{
  List *pLastElement = pRoot;
  List *pNewElement = NULL;
```

Tworzymy funkcję która będzie dodawała elementy do listy.
Tworzymy dwa wskaźniki, pierwszy z nich będzie wskazywał na ostatni element na liście, drugi będzie wskazywał na nowy element.

```
  while(pLastElement->pNext != NULL)
  {
    pLastElement = pLastElement->pNext;
  }
```

W pętli while, wyszukujemy ostatni element listy. Rozpoczynamy od elementu pRoot i szukamy ostatniego elementu (będzie on zawierał NULL w polu pNext).

```
  pNewElement = malloc(sizeof(*pNewElement));
  pNewElement->pNumRej = malloc(strlen(pNumRej) + 1u);
  strcpy(pNewElement->pNumRej, pNumRej);
  time(&(pNewElement->entryTime));
  pNewElement->pNext = NULL;
  pLastElement->pNext = pNewElement;
}
```

Tworzymy nowy element. Alokujemy miejsce w pamięci na element oraz na tablicę znaków o długości numeru rejestracyjnego który został podany w argumencie tej funkcji.
Następnie kopiujemy ciąg znaków, korzystamy z funkcji strcpy. UWAGA! Nie wolno skorzystać z operatora „=" przyrównania aby skopiować dwa ciągi znaków.

Aktualizujemy datę i godzinę wjazdu samochodu na parking.

Nowy element będzie ostatnim elementem w liście, dlatego wpisujemy NULL w pNext.

Następnie dołączamy nowy element do listy poprzez wpisanie jego adresu do pLastElement->pNext. Uwaga! pNewElement jest wskaźnikiem. Wywołując funkcję malloc, otrzymamy adres pamięci pod którym zaalokowane zostało miejsce na nasze dane.

Tworzymy funkcję zwracającą ilość samochodów na parkingu (elementów listy).
Tworzymy zmienną typu integer która będzie zliczać ilość elementów. Następnie w pętli while zaczynając od pRoot (pierwszego elementu) iterujemy po wszystkich elementach listy. Za każdym razem zwiększając wartość zmiennej size o 1. Na koniec funkcja zwraca ilość elementów.

```
int getNumOfCars(void)
{
  int size = 0;
  List *pLastElement = pRoot;
  while(pLastElement->pNext != NULL)
  {
    pLastElement = (pLastElement->pNext);
    size ++;
  }
  return size;
}
```

```
void removeCar(char *pNumRej)
{
   bool elementFound = false;

   List *pIterElementBefre = NULL;
   List *pIterElement = pRoot;
   List *pIterElementAfter = pRoot->pNext;

   while((NULL != pIterElement) && (false == elementFound))
   {
    if(0 != strstr(pIterElement->pNumRej, pNumRej))
    {
     elementFound = true;
    }
    else
    {
     if(NULL != pIterElement->pNext)
     {
        pIterElementBefre = pIterElement;
        if(NULL != pIterElement->pNext)
           pIterElementAfter = (pIterElement->pNext->pNext);
        else
           pIterElementAfter = NULL;
        pIterElement = (pIterElement->pNext);
     }
    }
   }
   if(true == elementFound)
   {
     if(NULL == pIterElementAfter)
     {
        pIterElementBefre->pNext=NULL;
     }
     else
     {
        pIterElementBefre->pNext=pIterElementAfter;
     }
     free(pIterElement->pNumRej);
     free(pIterElement);
   }
}
```

Funkcja usuwająca z listy wybrany element. Jako argument przekazujemy numer rejestracyjny pojazdu jako ciąg znaków.
Tworzymy trzy wskaźniki. Pierwszy wskazuje na element poprzedzający szukany element. Drugi będzie wskazywał na element poszukiwany. Trzeci będzie wskazywał na element kolejny (jeżeli istnieje).
W pętli while poszukujemy elementu którego numer rej jest równy numerowi podanemu w argimencie funkcji. Korzystamy z funkcji **strstr** która porównuje ciągi znaków.
Pętla while ma dwa warunki wyjścia:
- pierwszym jest test czy pIterElement istnieje. Jeżeli podany element nie istnieje to nigdy nie wykona się ciało pętli!
- drugim jest sprawdzenie czy poszukiwany element został odnaleziony.

Jeżeli funkcja **strstr** zwróci wartość „0” oznacza to że oba ciągi są takie same i odnaleziony został element. Jeśli wartość nie jest równa zero „przenosimy” się do następnego elementu w liście. Równocześnie aktualizujemy wskaźniki dla elementów **pIterElementBefore (n-1)** oraz **pIterElementAfter (n+1)**.

Jeśli element został odnaleziony następuje jego usunięcie.
Najpierw usuwamy dany element z listy oraz aktualizujemy łańcuch. Tak aby element **(n-1)** wskazywał na element **(n+1)**.
Sprawdzamy również czy usuwany element nie jest elementem ostatnim. Jeśli tak to wpisujemy NULL.

Następnie zwalniamy pamięć zaalokowaną na tablicę znaków przechowującą numer rejestracyjny elementu **n**(pITerElement->pNumRej).
Później zwalniamy pamięć zaalokowaną dla samego elementu **n (pITerElement)**.

UWAGA! Kolejność nie jest przypadkowa.
Jeżeli najpierw zwolnilibyśmy pamięć zapisaną pod adresem (pITerElement) to nie moglibyśmy dostać się do (pITerElement->pNumRej). Oznacza to, że ten kawałek pamięci zostałby „zgubiony”. Byłby ciągle zaalokowany, i nie moglibyśmy go zwolnić. Jest to tzw. wyciek pamięci.

```
int main(void)
{
 initList(); //Inicjalizujemy listę

 printf("Number of elements: %d\n", getNumOfCars());
 addCar("EL020GM");
 printf("Number of elements: %d\n", getNumOfCars());
 addCar("WY020GM");
 printf("Number of elements: %d\n", getNumOfCars());
 addCar("NA020GM");
 printf("Number of elements: %d\n", getNumOfCars());
 removeCar("WY020GM");
 printf("Number of elements: %d\n", getNumOfCars());
 addCar("RZ020GM");
 printf("Number of elements: %d\n", getNumOfCars());
 removeCar("EL020GM");
 printf("Number of elements: %d\n", getNumOfCars());

 deinitList(); //Konczymy prace z lista

 return 0;
}
```

Przykład wykorzystania opisanej listy.

Inicjalizujemy listę, następnie dodajemy do niej numery rejestracyjne kolejnych samochodów wjeżdzających na parking. W funkcji addCar, zostanie zapisany czas wjazdu na parking.
Usuwamy również pojazdy wyjeżdzające z parkingu w dowolnej kolejności.
Na konsoli wypisujemy listę samochodów na parkingu.

Na koniec wywołujemy funkcję deinitList() i kończymy działanie programu.